****
*   *
* DISKSCAN   *
*   *
* BY DAVID YOUNG *
*   *
****

DISKSCAN

BY DAVID YOUNG

>>>> THE ULTIMATE DISK UTILITY <<<<
FOR INSPECTING AND
CHANGING DISK DATA

MANIPULATE DISK DATA DIRECTLY
IN ANY OF 3 FORMATS:

1) Character
2) Hexadecimal
3) Assembly Language

MANY POWERFUL FEATURES INCLUDE:

SCREEN EDITOR - SCREEN DUMP
SEARCH DISK FOR SEQUENCE
FOLLOW SECTOR LINKS
HEX CONVERSION
MUCH MORE!

VERY FLEXIBLE AND USER FRIENDLY!

DISKSCAN USER'S GUIDE
by David Young

INTRODUCTION

DISKSCAN is a powerful and flexible disk utility made to run with the ATARI 810 disk drive. It is designed to include many features for advanced applications and yet be easy enough for a beginner to use.

In a nutshell, DISKSCAN turns the computer's screen into a window looking directly onto the floppy disk. It transforms the mysterious bit patterns of the disk into whatever format is convenient for the application, be it HEX data, ASCII characters or even disassembled object code! While viewing a sector, it can be altered by positioning the cursor over the byte to be changed and then typing the change. Alternatively, a line assembler can be used to modify 6502 object files. A special directory feature reveals the starting sector of any file while a search function can be used to search any file, or even the whole disk, for a 1 or 2 byte sequence. Large groups of sectors can be imaged to another disk, transformed into a binary load file or even dumped to a printer. All of DISKSCAN's functions employ a user interface designed to anticipate responses and detect errors.

Since DISKSCAN is menu driven, users already familiar with the structure of a disk can begin using the program immediately. At some point, however, it would be wise to read the detailed descriptions of the individual commands in order to become privy to their more subtle features. Those desiring to learn more about the disk should begin with the tutorial covering the different types of data structures to be found on the disk. This is not quite as scary as it sounds because there are not that many different types of data and the interactive use of DISKSCAN makes it all quite painless, even fun! Once the use of DISKSCAN is mastered it will become an indispensable tool to any programmer with a disk based ATARI PERSONAL COMPUTER.

SYSTEM REQUIREMENTS

ATARI 800/400 Personal Computer
ATARI 810 Disk Drive
24K RAM (32K for Assembler/Disassembler option)
Printer (optional)

GETTING STARTED

NOTE: Both sides of the DISKSCAN disk have a special format. Although the DISKSCAN programs can be backed up to other disks, they will run only if the original DISKSCAN disk is in drive 1. If the front side becomes defective, the backside serves as a backup. DO NOT REFORMAT EITHER SIDE OF THE ORIGINAL DISK!

1. Turn on your disk drive and insert the DISKSCAN diskette. Insert the BASIC cartridge and power up your computer.

2. When the READY prompt displays, type RUN "DISKSCAN.BIG" for a 32K computer or RUN "DISKSCAN.SML" for a 24K computer.

When the DISKSCAN menu appears, remove the DISKSCAN diskette and insert the diskette to be "scanned" into the drive.

THE MENU FUNCTION: M

The first thing that appears on the screen when DISKSCAN is run is the menu. It consists of a list of letters which represent the primary functic available to you along with a short description of each function. In order to call up a function, you simply enter the letter which represents it when asked to do so. If you forget which function certain letters represent, then you can always return to the menu by using the "M" function. Since the function letters are supplied each time a function is requested, you will not find it necessary to return to the menu very often once you have gained a familiarity with DISKSCAN.

Associated with the menu are three questions. Your answers to these questions let DISKSCAN know how the sectors of a file are related to each other (either sequentially or linked), how you would like the sector data to be represented on the screen (either in ATASCII characters or hexadecimal format), and in which drive the disk being scanned is. To change the sector mode, the screen format or the drive #, you can return to the menu with the "M" function. It is also possible (and more convenient) to change the screen format with the "T" (toggle screen format) function. If you are unsure how to answer these questions, it is highly recomended that you read the tutorial on disk data structures before proceeding.

THE FUNCTION PROMPT

To call up a DISKSCAN function, enter the letter which represents that function when the following prompt appears on the screen:

```
R,W,S,C,D,B,I,X,A,T,G,P,H or M?
```

This is referred to as the "function prompt". It consists of a list of the available functions followed by a question mark. Type the letter of the desired function but DO NOT follow it by RETURN. At this point a sector number may be requested. If so, supply the sector number either in decimal or hex (e.g., 20 or $14) and hit RETURN. The function will then be executed. If you cannot remember which letter corresponds to a certain function, use "M" to return to the menu.

INPUT FORMATS

In the interest of user friendliness, DISKSCAN will accept put in four different formats: decimal, hexadecimal, character, and sembly language. Decimal and hex are valid anywhere a number is expected. Characters may be entered if DISKSCAN is in the character mode and you are using either the change ("C") or scan ("S") functions. Assembly language is used in conjunction with the assembler function.

To input a hex number, precede it with "$". Thus, in response to "Sector #?", you could enter either "$1ØØ" or "256" with the same result. The only exception to this rule is in the assembler, where most numerical operands MUST be specified in hex but need not be preceeded by "$". The only assembler instructions that must have decimal operands are the conditional branches. The relative displacement MUST be specified as a decimal number preceded by "+" or "-". The reason for these conventions is that the output of the disassembler, which conforms to the same rules, is limited to the right hand margin of the screen. This format actually turns out to be quite convenient for most assembly language applications.

RESPONDING WITH RETURN

When DISKSCAN is awaiting your input, hitting RETURN preceded by no other characters is always a valid response. The program tries to interpret it in a manner convenient to you. For instance, when the function prompt appears, hitting RETURN will cause DISKSCAN to assume the previous function. In response to "Sector #?", a RETURN will cause the program to increment to the next sector if the command was the same, or it will assume the same sector if the command just changed. Within certain contexts, RETURN causes the function to be aborted.

If, for example, you would like to read quickly through several sectors, you would enter "R" in response to the function prompt and then supply the number of the first sector. From then on, if you enter only RETURNs in response to the function and sector prompts, DISKSCAN will continue to read the sectors of a file one after another. This is especially useful if the program is in linked mode as it will automatically calculate the next sector from the link at the end of the current sector.

DIRECTORY OF THE DISK: D

The directory function is useful for determining the location of a file on a DOS disk. (This function is only valid for disks with DOS formats.) It will read the 8 sectors of the directory one at a time and list the filenames of the directory entries along with with the file sizes and, most importantly, the starting sector of each file (all in decimal). It will stop after reading each sector and ask you if you have found the information that you needed. If you answer with anything other than "Y", it will read the next sector of the directory. If you answer with "Y" or if there are no more directory entries, the function prompt will reappear awaiting your next command.

To execute the directory function, type "D" when the function prompt appears. The first sector of the directory will be read and displayed immediately.

READ SECTOR: R

You will probably use this function of DISKSCAN more than any other. It reads an entire sector (128 bytes) from disk and displays its contents either in hex or as ATASCII characters, depending on the current screen format. The 128 bytes are arranged on the screen in a matrix of 8 columns by 16 rows. The address of the the first byte of each row is given in hex in the left margin. The sector number is provided at the top of the screen in both decimal and hex.

To execute the read function type "R" when the function prompt appears. Then provide the sector number either in decimal or hex (preceded by "$") and hit RETURN. The desired sector will be read and displayed immediately. The function prompt will then reappear awaiting your next input.

TOGGLE SCREEN FORMAT: T

This function is useful for changing the format of the screen output from hex to character or vice versa without having to return to the menu. In addition, it actually reprints the screen into the new format at the same time. Thus, if you were using the character format, you could type "T" to observe the sector link in hex and then flip back to character format by typing "T" again.

To execute the screen toggle function, type "T" when the function prompt appears.

PRINT SCREEN CONTENTS: P

This function dumps the contents of the screen to the printer. It does not matter what is being displayed at the time except that, if the screen is in character format, certain characters are unprintable and are printed as dashes. Also note that inverse video characters are printed as normal characters. Thus, you can conveniently make a hard copy of sector data, the menu, or the disk directory. This is especially useful if you are going to alter a sector and you need a record of its original contents.

To execute the screen dump function, type "P" when the function prompt appears.

HEX CONVERSION: H

The hex conversion function is a convenient way to convert hexadecimal numbers to decimal and vice versa. It will operate on any number from Ø to 65535 ($Ø to $FFFF). If you use the assembler or disassembler much this function may come in handy.

To execute the hex conversion function, type "H" when the function prompt appears. Then type the number to be converted and hit RETURN. Remember, always precede a hex number by "$". Both the decimal and hexadecimal form of the number will appear just above the function prompt and will remain there until the execution of another function causes it to be erased.

CHANGE CURRENT SECTOR: C

The change sector function is one of the most powerful features of DISKSCAN and yet it is extremely easy to use. A screen editor is lemented, meaning you simply position the cursor over the desired e of the sector and type the change.

This function is usually preceded by the "R" (read sector) function whereby a sector is read from disk into memory. When you use the change function you are really only altering the image of the sector in memory, not the actual disk sector. The sector on the disk will be updated only if you use the "W" (write sector) function to copy the memory image back to disk.

The cursor is positioned with the normal ATARI cursor control keys: CTRL-, CTRL=, CTRL+, and CTRL* (the little arrows on these keys will not print here). Notice that, when the cursor reaches the end of the line, it will automatically wrap to the beginning of the next line. Likewise, if you try to go below the last line of the sector it will wrap to the top. Thus, since the cursor starts in the top left corner of the sector, the easiest way to get to the end of the sector is to move the cursor to the left. You will probably have to try this out to see how it works. At any rate, it is impossible to move the cursor off of the sector matrix.

When you have the cursor positioned over the byte you desire to change, type the new byte over the old. This will require 1 keystroke per byte when the screen is in character format or 2 keystrokes per byte in the hex format. Notice that the cursor automatically moves over as you type the change. Thus, you can alter any number of successive bytes without having to reposition the cursor. When in hex format, only valid hex characters are allowed. If you try to type any other characters, the sector buffer in memory l not be altered but the buzzer will sound and a blank will be left in that bble position, indicating you need to go back and correct it. However, when in character format, ALL keystrokes print their ATASCII representations except the cursor controls and ESC, which is used to exit the change function. Of course, BREAK is also not allowed unless you want to suspend execution of DISKSCAN. If you should accidently hit BREAK, clear the screen and type RUN. Your sector will still be in memory.

To execute the change function, type "C" when the function prompt appears. Position the cursor and type the change. Exit the screen editor with ESC. Now write the sector back out to disk with the "W" (write sector) function if you desire to save it (see next section).

IMAGE SECTORS: I

This function is used to make an exact copy of one or more sequentially numbered sectors to another disk. The imaged sectors will retain their same numbers on the destination disk. While this function could be used to copy an entire disk, its main purpose is to allow you to easily keep your backup current as you modify a disk. In other words, at convenient points during modification of a disk you should make images of the sectors you have been working on to a secondary backup disk (not to the original backup). Of course, the read and write functions ("R" and "W") could just as easily be used if you only need to copy 1 or 2 sectors.

The number of sectors you can image at one time is limited by the amount of free memory space available. Thus, the more RAM you have the better. However, there is another way to make more RAM available to the image function. If you have at least 32K of RAM then you will normally want to run DISKSCAN.BIG because it has the assembler/disassembler functions available. DISKSCAN.SML is the shorter implementation of DISKSCAN, made to run out of at least 24K. Thus, even if you have more than 24K of RAM, you may use DISKSCAN.SML to make available to the image function the maximum free memory space. This same discussion applies to the binary load file function ("B") as well.

To execute the image function, type "I" when the function prompt appears. Supply the starting sector number and hit RETURN. You will then be informed of the maximum number of sectors you can image at one time. Supply the number of sectors you wish to image and hit RETURN. When prompted, insert the destination disk and hit RETURN. When the function prompt appears, the image operation is complete and you may reinsert the source disk.

BINARY LOAD FILE: B

This function is used to make a binary load file out of one or more sectors. That is, you can make a DOS file out of the sectors of, say, a ga boot disk. Then you can load the file into memory with the binary load option of DOS and perhaps disassemble it or even execute it. It will work in either sector mode, sequential or linked, but it will usually only make sense in the sequential mode. Though this is a very powerful function, it will be useful only to advanced users.

The number of sectors you can make into a binary load file at one time is limited by the amount of free memory space available. Thus, the more RAM you have the better. However, there is another way to make more RAM available to the binary load file function. If you have at least 32K of RAM then you will normally want to run DISKSCAN.BIG because it has the assembler/disassembler functions available. DISKSCAN.SML is the shorter implementation of DISKSCAN, made to run out of at least 24K. Thus, even if you have more than 24K of RAM, you may use DISKSCAN.SML to make available to the binary load file function the maximum free memory space. This same discussion applies to the image sectors function ("I") as well.

To execute the binary load file function, type "B" when the function prompt appears. Supply the starting sector number and hit RETURN. You will then be informed of the maximum number of sectors you can make into a binary load file at one time. Supply the desired number of sectors and hit RETURN. When prompted, insert the destination disk (make sure it is a DOS disk), supply the load address to be appended at the beginning of the file, and hit RETURN. When the function prompt appears, the binary load file is complete and you may reinsert the source disk.

ASSEMBLY LANGUAGE SU

Working in hex or character format is fine
for inspecting and patching 6502 machine la
port is almost a necessity. To this end, DI
assembler and disassembler, but also a unique "
for locating specific addresses within a binary
facilities are only included in DISKSCAN.BIG fo
with at least 32K of RAM. There are several bo
6502 assembly language. One that I recommend i
Software for Your 6502 personal computer by Ker

DISASSEMBLE A SECTO

The disassembler function can translate the
binary file into standard assembly language ins
disassemble, outputting to the right margin of
you specify within a sector until the end of th
will continue disassembling with the next logic
current sector mode: sequential or linked. A m
between 2 sectors is handled gracefully with no
in the sector data matrix indicate the region o
disassembled.

The output of the disassembler is designed
to the right of the sector data matrix. For th
operands are specified in hex, but without the
distinguishes hexadecimal from decimal. The or
which have decimal operands are the conditional
ways preceded by "+" or "-" to indicate the d
splacement. To find the destination of the b
of the next instruction and count forward or ba
bytes indicated by the displacement.

Several other DISKSCAN functions compliment
need a hard copy of the disassembler output, us
hexadecimal operand can be converted to decimal
need to find a specific address within a machin
function.

To execute the disassembler function, type
prompt appears. The current sector will be dis
prompted for the starting byte. Since the byte
to the left of tne sector matrix, the starting
in hex (preceded by "$"). The disassembly will
point until the right margin is full. When the
enough, any response other than "Y" will cause
This is also the case when the end of the secto
to the prompt will cause the disassembler to be
function prompt to appear.

ASSEMBLY LANGUAGE SUPPORT

Working in hex or character format is fine for many DISKSCAN applications, for inspecting and patching 65Ø2 machine language files, assembly language port is almost a necessity. To this end, DISKSCAN not only offers an assembler and disassembler, but also a unique "GOTO binary address" function for locating specific addresses within a binary file. The assembly language facilities are only included in DISKSCAN.BIG for running on machines with at least 32K of RAM. There are several books on the market which cover 65Ø2 assembly language. One that I recommend is Beyond Games: Systems Software for Your 65Ø2 personal computer by Ken Skier.

DISASSEMBLE A SECTOR: X

The disassembler function can translate the 65Ø2 machine code of a binary file into standard assembly language instructions. It will disassemble, outputting to the right margin of the screen, from the point you specify within a sector until the end of the sector. If you desire, it will continue disassembling with the next logical sector, predicated on the current sector mode: sequential or linked. A multibyte instruction split between 2 sectors is handled gracefully with no discontinuity. Little arrows in the sector data matrix indicate the region of the sector being disassembled.

The output of the disassembler is designed to fit into the limited space to the right of the sector data matrix. For that reason, most numerical operands are specified in hex, but without the preceding "$" which usually distinguishes hexadecimal from decimal. The only disassembler instructions which have decimal operands are the conditional branches and these are ways preceded by "+" or "-" to indicate the direction of the relative splacement. To find the destination of the branch, start at the beginning of the next instruction and count forward or backwards the number of bytes indicated by the displacement.

Several other DISKSCAN functions compliment the disassembler. If you need a hard copy of the disassembler output, use the "P" function. A hexadecimal operand can be converted to decimal with the "H" function. If you need to find a specific address within a machine language program, use the "G" function.

To execute the disassembler function, type "X" when the function prompt appears. The current sector will be displayed and you will be prompted for the starting byte. Since the byte addresses are provided in hex to the left of tne sector matrix, the starting byte is most easily provided in hex (preceded by "$"). The disassembly will proceed from that point until the right margin is full. When the program asks if that is enough, any response other than "Y" will cause the disassembly to continue. This is also the case when the end of the sector is reached. Answering "Y" to the prompt will cause the disassembler to be terminated and the function prompt to appear.

ASSEMBLE INTO SECTOR: A

When modifying 65Ø2 machine language, a convenient alternative to the change function ("C") is the assembler function. It implements a line assembler (meaning that each instruction is assembled as you type it in) to modify the current sector in memory. It accepts standard 65Ø2 assembly language instructions and will beep at you if you attempt to input something illegal. You can start assembling anywhere within the sector (except within the sector link if in linked mode). When the end of the sector is reached, if desired, the current sector will be written out and the next logical sector (predicated on the current sector mode: sequential or linked) will be read in. A multibyte instruction split between 2 sectors is handled gracefully with no loss of data.

The numerical operands of the assembler conform to the same format as the disassembler. All numerical operands, except the displacements of conditional branches, must be given in hex and may or may not be preceded by "$". The relative displacements of conditional branches must be specified in decimal preceded by "+" or "-" to indicate the direction of the branch. To calculate this displacement, start at the first byte of the following instruction and count forward or backwards the number of bytes to the destination address.

Should any question arise as to how to specify any of the 13 65Ø2 addressing modes, study the output of the disassembler. Leading zeros and imbedded blanks are ignored by the assembler.

To execute the assembler function, type "A" when the function prompt appears. Provide the starting byte (most conveniently in hex preceded by "$") and hit RETURN. You may then begin entering assembly language instructions followed by RETURN. Each instrucion will be translated into machine code and inserted into the sector buffer, overwriting the current contents. A little arrow in the sector matrix indicates the point at which the next instruction will be inserted into the sector. If the end of the sector is reached, you can cause the current sector to be written out and the next logical sector to be read in by responding to the prompt with "Y". Any other response causes the assembler to be terminated. The assembler can also be terminated by hitting RETURN when prompted for the next instruction. Upon termination, the function prompt will reappear.

GOTO BINARY ADDRESS: G

Inspecting a binary file can be very tedious if the flow of the program ps around much. This is because it is very time consuming to calculate ch byte of which sector is referenced by an absolute address. In fact, if the task at hand is a large one, it is recommended that you use the "B" function to create a binary load file out of the sectors of the program (unless, of course, it already is a binary load file with a convenient load address). This would allow you to load the program into memory for processing by a full blown disassembler. However, for casually inspecting a binary file, the GOTO function is a good way to find your way around. It allows you to jump to any absolute address within the file without doing a single calculation.

When using the GOTO function, you can take advantage of the fact that the load address is usually located at the very beginning of a program. After you have specified this as the base address then the GOTO function can calculate the exact location on the disk of any absolute address within that program. It bases its calculation on either 125 bytes/sector, if the sectors are linked, or 128 bytes/sector, if they are sequential. Once the address is found, the disassembler ("X") can be executed starting at that location.

To execute the GOTO function, type "G" when the function prompt appears. Then provide the value, sector, and byte number of the base address as they are requested. If you have already entered these once, just hit RETURN at the first prompt. You will then provide the destination address. When you hit RETURN, the program will go find the address, display the sector and print the byte number of the address above the function prompt so that it can be referenced when choosing the next function. Remember, if the base address remains the same, it is not necessary to specify it after the rst use of the GOTO function.

QUESTIONS?

I have tried to make this program as powerful and user friendly as possible. I would appreciate any comments or suggestions. Also, if you have any questions, feel free to write.

David Young
421 Hanbee
Richardson, TX
75080

ATARI Disk Data Structures: An Interactive Tutorial

by David Young

INTRODUCTION

The floppy disk is a marvelous and yet mysterious media for mass storag of data. Indeed, to understand exactly how a bit of data is stored and retrieved from the surface of the disk requires a good knowledge of physics. However, to learn about the data structures found on a disk requires no higher mathematics than hexadecimal arithmetic. The manual supplied with the computer usually does an adequate job of supplying all the technical details, but wouldn't it sink in better if the actual data on the media were viewed while it is being described? The DISKSCAN program is used to demonstrate the disk data structures as they are being described. Follow the instructions under GETTING STARTED in the DISKSCAN USER'S GUIDE to run DISKSCAN. Once the program has started, remove the DISKSCAN diskette and insert some other diskette that has been backed up. Use the "R" (read sector) function whenever you are requested to view a particular sector on disk. Whenever you are requested to change the display format from hex to character, or vice versa, use the "T" (toggle display format) function.

The Disk Media

The first disk structure to be aware of is the sector, which on any computer system consists of a group of contiguous bits recorded at a specific location on the disk. The disk drive hardware always operates on whole sectors, that is to say, it is not possible to read or write partial sectors. Groups of sectors are organized into tracks forming concentric rings about the center of the disk. The ATARI system divides the disk into 4Ø tracks with 18 sectors per track for a total of 72Ø sectors. T s is best visualized by taking the lid off of the disk drive and watching the read/write head move as certain sectors are addressed. On the ATARI 81Ø dis drive this is accomplished by removing the 4 phillips head screws hidden under gummed tabs at each corner of the lid. While inside the case, a bit of lubrication on the 2 cylindrical guide rails supporting the head will make the drive less noisy.

If sectors 1 through 18 are read with DISKSCAN, the head remains fixed on the outermost track. When sector 72Ø is read, the head moves in to the innermost track. When a disk is formatted, the head can be seen to bump sequentially through all 4Ø tracks. It is laying down the patterns on the oxide surface which will be recognized by the drive hardware as the sectors. The sectors are all initially empty (128 bytes of Ø), but at the end of the formatting routine, as described in the next section, the ATARI DOS records special data into certain sectors. The top of the drive can now be resecured. No more information about the hardware is needed to understand the higher level disk data structures of the software.

Boot Sector

At the end of the formatting process DOS reserves and initializes certain sectors for special tasks. Into sectors 1 through 3 is stored the bootstrap for DOS. On power-up the ATARI operating system reads sector 1 to determine how many sectors to read and where into memory to load them. After it has loaded in the specified number of sectors, DOS starts executing the new code at the load address + 6. Put DISKSCAN into the hex mode and read sector 1 of any DOS disk. Byte 1 says that 3 sectors are read (sequentially) and bytes 2 and 3 specify a load address of $7ØØ. (A 2 byte number is ays specified with the least significant byte first.) Byte 6 is the st intruction to be executed (a $4C14Ø7 is a JMP $714). In this case the code which follows sets up to load the File Management System of DOS into memory. This is called the second stage of the boot. Look at the first sector of any other boot disk available (any game or program which loads in from disk on power-up). It might be seen that the program loads in entirely during the first stage of the boot, i.e. byte 1 of sector 1 has a sector count which represents the entire program. For more details on the disk boot process, see the ATARI Operating System User's Manual.

Volume Table of Contents

Besides the first three boot sectors, DOS sets up sectors 36Ø to 368 as the directory of the disk. DOS uses the directory to keep track of where files are stored on disk and how much disk space remains. Read sector 36Ø of a DOS disk with DISKSCAN in the hex mode and view a part of the directory called the Volume Table of Contents (VTOC).

Information pertaining to the availability of every sector on the disk is stored in this sector. Bytes 1 and 2 specify the maximum number of user data sectors on the disk ($2C3 = 7Ø7) and bytes 3 and 4 specify the number of free sectors remaining on the disk (7Ø7 for an empty disk, Ø for a full one). Starting in bit 6 (the second to highest order bit) of byte $ØA, each bit up through byte $63 corresponds to a sector. A 1 corresponds to a free sector while a Ø means the sector is being used.

When a file is stored on the disk, the bits corresponding to the sectors d are set to Ø. When the file is erased, the bits are set back to 1. That is why DOS, when it deletes a file, can be heard reading the entire file. It is determining which sectors were being used by the file so that it can free them back up. Notice that bits 1, 2 and 3 (bits 6, 5 and 4 of byte $ØA) are set to Ø. These correspond to the 3 boot sectors. Likewise, the 9 bits starting in byte $37 are Ø because they correspond to the sectors of the directory. These 12 sectors are thus kept from being overlaid by user files.

If the VTOC is viewed on an older disk which has had many file additions and deletions, it may be noted that the VTOC has become quite fragmented. Any file added to the disk may get stored into sectors scattered about the disk. How DOS keeps track of files spread over multiple sectors will be discussed shortly. By the way, even though the operating system recognizes sector 72Ø (try reading it; should be all zeroes), DOS never makes use of it. True to Murphy's Law, it adopted the number scheme of Ø to 719 instead of 1 to 72Ø. No need to bother trying to read sector Ø!

The Directory

Of all the disk data structures, probably the most important one to be acquainted with is the directory. The 8 sectors following the VTOC (361-368) contain a list of all the files on the disk along with their size, starting sector and status. Put DISKSCAN into character mode and read sector 361 of a

DOS disk that has several files on it. It can be seen that the name of the first file starts in byte $Ø5 and the extension (if any) starts in byte $ØD. If any of the 11 character positions of the filespec are unused, it contains a blank. Notice that the filenames start every 16 bytes, allowing 8 directory entries per 128 byte sector. Thus, the maximum number of entries for the 8 sectors of the directory is 64.

Now put DISKSCAN in hex mode and read sector 361. The first byte of each 16 byte entry contains the status of the file. For a normal file that byte is $42, unless it is locked, in which case it has a status of $62. A deleted file has a status of $8Ø. An anomaly occurs whenever a file is opened for output (from BASIC, perhaps) but is not closed before the computer is powered down or glitched. Since the status of an open file is $43, DOS will neither recognize the entry as "in use" nor "deleted". Even the sectors which may have been written out will not really exist on disk because the VTOC is not updated until the file is closed. The only harm done is that this bogus entry will take up space in the directory until the disk is reformatted. (One other solution would be to change the $43 to an $8Ø using DISKSCAN; refer to the change sector function, "C", in the DISKSCAN USER'S GUIDE.) The second and third bytes of each entry contain the size in sectors of the file (low order byte first) while the fourth and fifth bytes specify the first sector of the file. DOS only needs to know the first sector of a file because each sector points to the next sector of the file in a process called "linking".

Linking

At this point it would be best to explain how DOS forms a data file on disk. First, the user must open an I/O channel for output to the disk, perhaps with the BASIC "OPEN" command. DOS responds by creating an entry in the directory with the specified filename and a status of $43. DOS reads the VTOC into memory and searches the disk map for the first free sector. If free sector is found, it's number is used as the starting sector in the directory entry. Now, when the user begins to output data via this I/O channel, perhaps with the BASIC "PUT" command, DOS waits until it has collected 125 bytes of user data in a buffer. Then DOS adds 3 special bytes of it's own and outputs the sector to the disk. I call these 3 bytes the "sector link".

The sector link, bytes 125 to 127 of the sector, contains 3 pieces of information. The high order 6 bits of byte 125 contain a number which represents the position of the file's entry within the directory (Ø to 63). DOS uses this number to check the integrity of the file. If ever this
ity of the file. If ever this
number should fail to match the position of the file's directory entry, DOS generates an error. The low order two bits of byte 125 and all of byte 126 form a pointer to the next sector of the file. A pointer is the address of a record in the computer's memory or, in this case, the address of a record on disk, the sector number. The next sector of the file is determined by scanning the bit map of the VTOC for the next free sector, which may or may not be the next sequential sector of the disk. Thanks to the link pointers, all sectors of a file need not be contiguous sectors on the disk. The last byte of the sector link (byte 127 of the sector) contains the number of bytes used within the sector. This byte will always be $7D (125) except for the last sector of a file, which will probably be only partially filled. DOS writes out this partial sector only when the user closes the file, perhaps with the BASIC "CLOSE" command.

When an output disk file is closed, DOS writes the newly updated VTOC back out to sector 36Ø. It then updates the file's directory entry by changing the status to $42 and filling in the file size (bytes 1 and 2) with the number of sectors used by the file. This completes the process of creating a file on disk. Now, when DOS is requested to read a file from disk, it finds the directory entry of the specified file to determine the start sector. Then, following the link pointers, it reads the file sector by sector until EOF (end of file) is reached, indicated by a link pointer of Ø.

Equipped with a basic understanding of how a file is stored on disk, try looking at a file with DISKSCAN. In character mode, first locate the name of the desired file in the directory (sectors 361-368). Then put DISKSCAN in hex mode and look at the fourth and fifth byte of the entry to determine the start sector. For example, if these two bytes were "1F Ø1", type "$1ØF" in response to "Sector #?" to read the first sector of the file. Observe the last three bytes of the sector and verify that the high order 6 bits of byte 125 correspond to the directory entry position and that byte 127 is the number of bytes used (probably $7D). Then determine the next sector of the file from the low order 2 bits of byte 125 and byte 126. For example, if bytes 125 and 126 are "1D 2Ø" then the next sector of the file is $12Ø and the file is the eighth entry of the directory (the first entry being entry Ø). If the file is not too long, it would be instructive to follow the sector links to EOF. Once the ability of finding a file on disk and following the sector links is mastered, all that remains is to become familiar with the 3 types of files used by DOS. (NOTE: DISKSCAN will automatically follow the sector links of a file if it is in linked mode; refer to the DISKSCAN USER'S GUIDE.)

File Types

The first type of file is not a true file, per se, because there is no entry in the directory for it. This file type includes the boot record and the directory itself. And since the sectors which make up these files are not linked but, instead, are related to each other sequentially, I call these records "sequentially linked files". When examining a sector of the boot record or directory, merely increase the sector number by 1 to get to the next sector of the record.

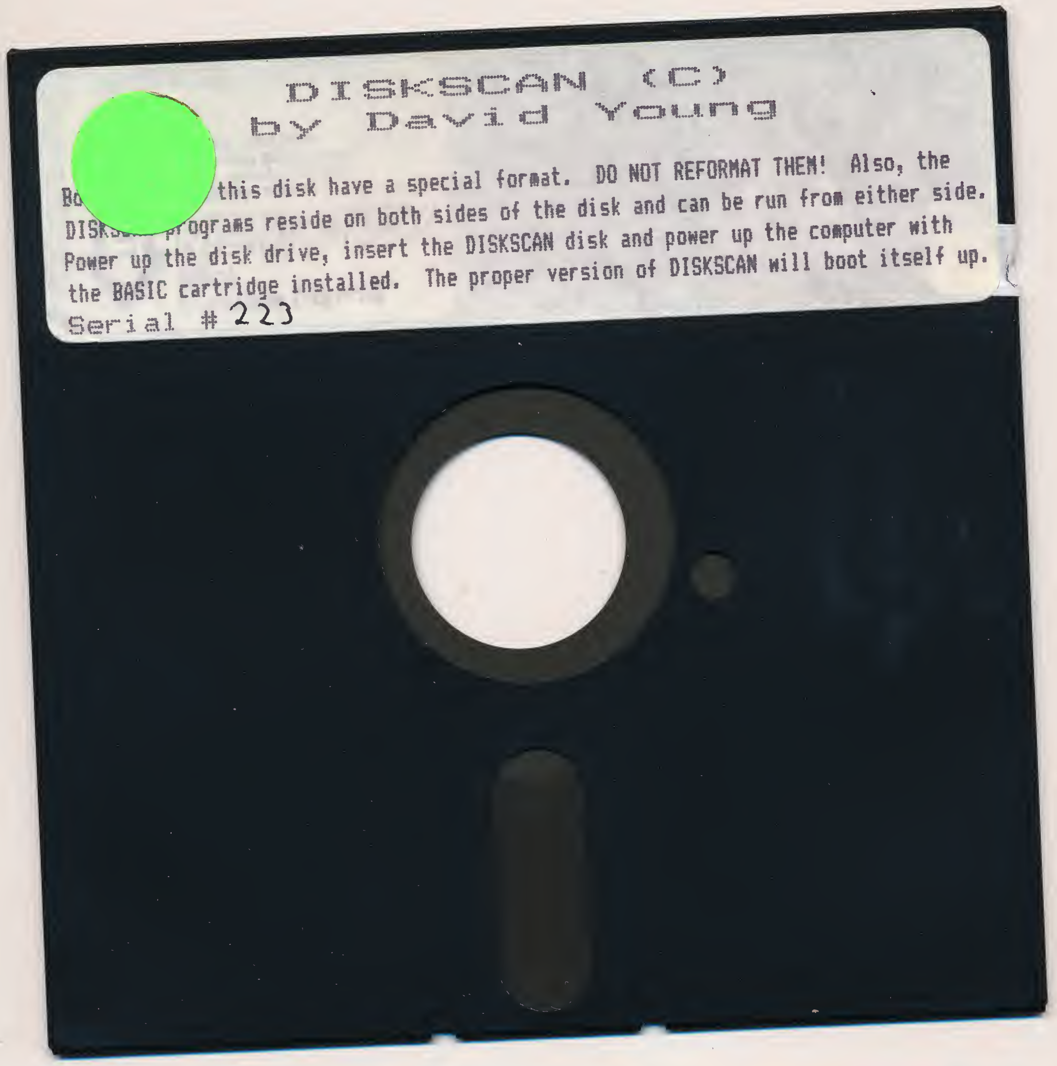
An example of the second type of file is that which is created with the BASIC "LIST" or "SAVE" command. This file consists of ASCII characters which either represent straight text, as in a LISTed file, or a sort of condensed text, as in a tokenized or SAVEd file. Except when viewing the sector links, the character mode of DISKSCAN is best suited for examinimg this type of file. At this point it would be instructive to locate in the directory of a DOS disk a file created with the BASIC "LIST" command. Upon determining the start sector, observe the file in the character mode. The BASIC program can be easily recognized. It may be noted that the carriage return-line feed character (CRLF) is displayed in it's ATASCII representation (an inverse escape character) instead of executed. Now observe a file that consists of a program that was SAVEd from BASIC. Since the text has been tokenized the program is harder to recognize. However, certain parts of the program are not altered during the tokenization process, notably text following REM and PRINT statements. Now, having investigated ASCII files, it is time to discuss the last file type, the "binary load" file.

The binary load file is primarily used to load 65Ø2 machine code into memory for execution. However, it's format is so general that it can be used just as easily to load any type of data, including ASCII text. Locate a game or other program which is run with the BINARY LOAD option of DOS. Alternatively, create a binary load file by saving any part of memory (except ROM) with the BINARY SAVE option. Now observe the first sector of the file with DISKSCAN in the hex mode. First, notice that all binary load files start with 2 bytes of $FF. The next four bytes are the start and end addresses, respectively, where the data to follow will be loaded into memory. If these four bytes were "ØØ AØ FF BF" then the data would be loaded between the addresses of $AØØØ and $BFFF. I call these four bytes a load vector. After DOS has loaded in enough bytes to satisfy the load vector, it assumes, unless EOF is reached, that the next four bytes specify another load vector. DOS will continue inputting the file at this new address.

Upon completion of a BINARY LOAD, control will normally be passed back to the DOS menu. However, DOS can be forced to pass control to any address in memory by storing that 2 byte address at location $2EØ. To store the 2 bytes, it is necessary to specify another load vector as part of the file. If, for example, it were desired to execute the program loaded in at $AØØØ, the following load vector would be part of the file: EØ Ø2 E1 Ø2 ØØ AØ. I call this specialized load vector an autorun vector. It achieves the same result as the RUN AT ADDRESS option of DOS. Try to find the autorun vector in the file being viewed. Although it could be at the beginning, it is most likely located at the very end of the file.

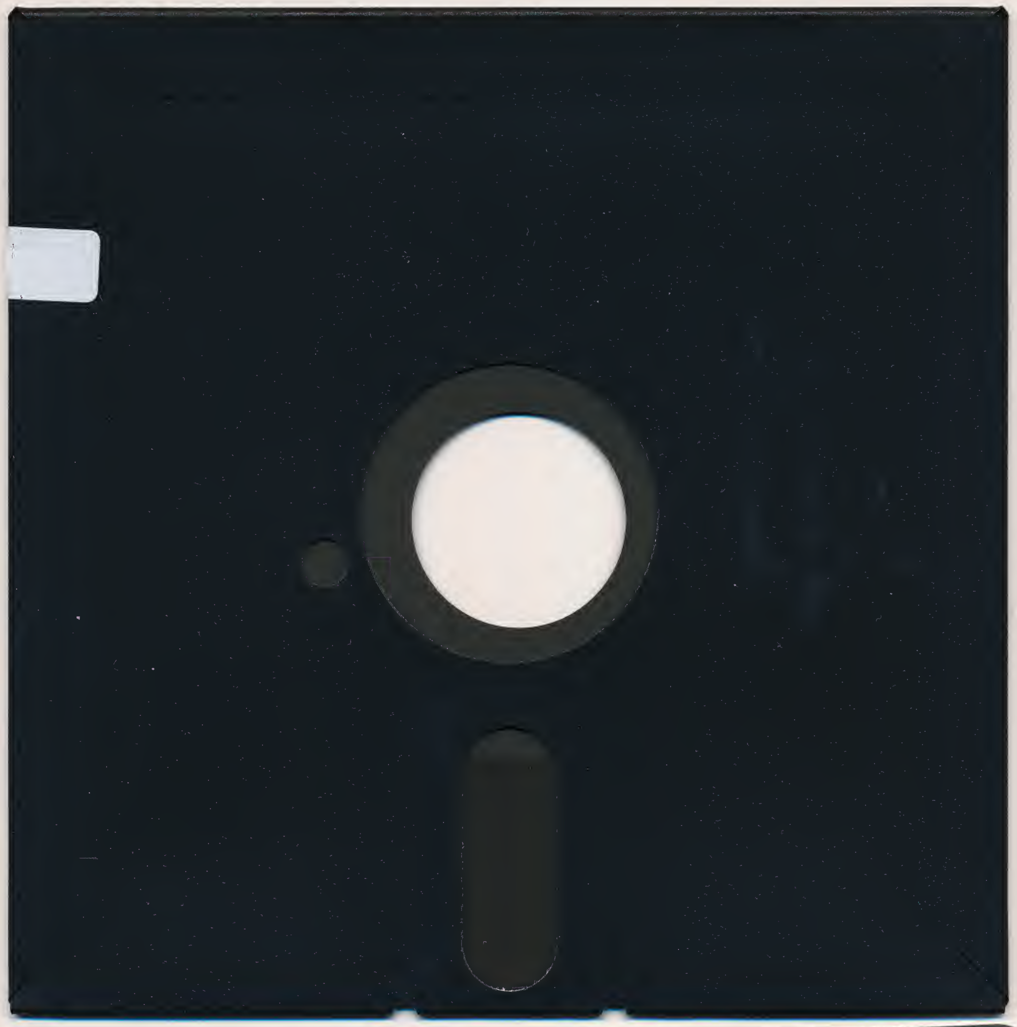
Conclusion

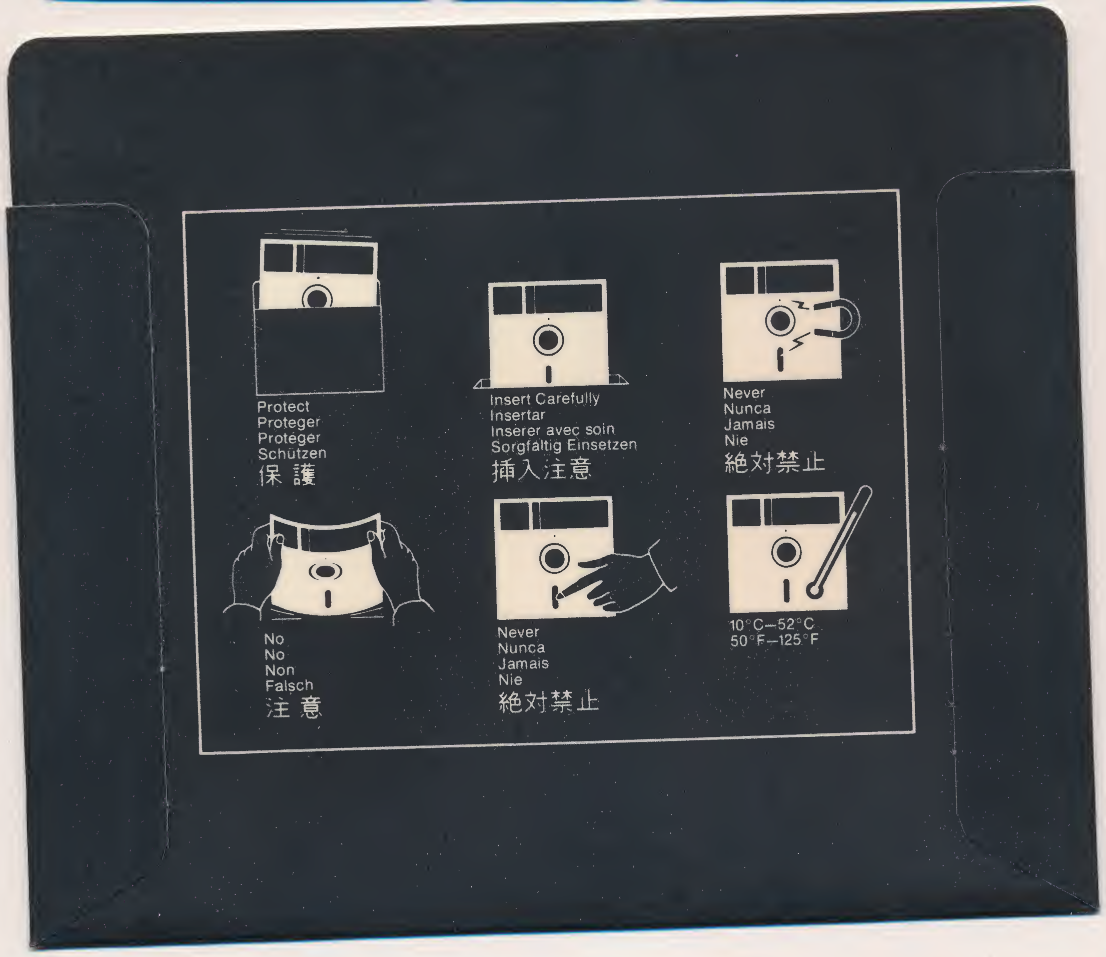
Anyone who has made it to the end of this tutorial and has successfully performed the exercises presented here should consider themselves proficient on the subject of ATARI disk data structures. I hope this tutorial has been useful to those wishing to gain a perspective about how data is stored on disk. At the very least it should have taken some of the mystery out of working with this popular device.



DISKSCAN (C)
by David Young

Bo this disk have a special format. DO NOT REFORMAT THEM! Also, the
DISK programs reside on both sides of the disk and can be run from either side.
Power up the disk drive, insert the DISKSCAN disk and power up the computer with
the BASIC cartridge installed. The proper version of DISKSCAN will boot itself up.
Serial # 223





Protect
Proteger
Proteger
Schützen
保 護
Insert Carefully
Insertar
Inserer avec soin
Sorgfaltig Einsetzen
挿入注意
Never
Nunca
Jamais
Nie
絶対禁止
No
No
Non
Falsch
注 意
Never
Nunca
Jamais
Nie
絶対禁止
10°C–52°C
50°F–125°F